37 | 从内核到应用：网络数据在内核中如何流转

2021-08-02 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 13:27 大小 12.33M

你好，我是 LMOS。

上节课我们对一次请求到响应的过程积累了一些宏观认识，相信你已经对整个网络架构有了一个整体蓝图。这节课，让我们来仔细研究一下网络数据是如何在内核中流转的，让你开阔视野，真正理解底层工程的实现思路。

凡事先问目的，在网络数据在内核中的流转，最终要服务于网络收发功能。所以，我会先带你了解一次具体的网络发收过程，然后带你了解 lwIP 的网络数据收发。有了这些基础，我还会示范一下如何实现协议栈移植，你可以在课后自行动手拓展。

好，让我们正式开始今天的学习吧。课程配套代码，你可以点击这里获取。

先看看一次具体的网络发收过程

理解软件的设计思想，最重要的是先要理解需求。而内核中的数据流转也只是为了满足网络收发的需求而进行的设计。

发送过程总览

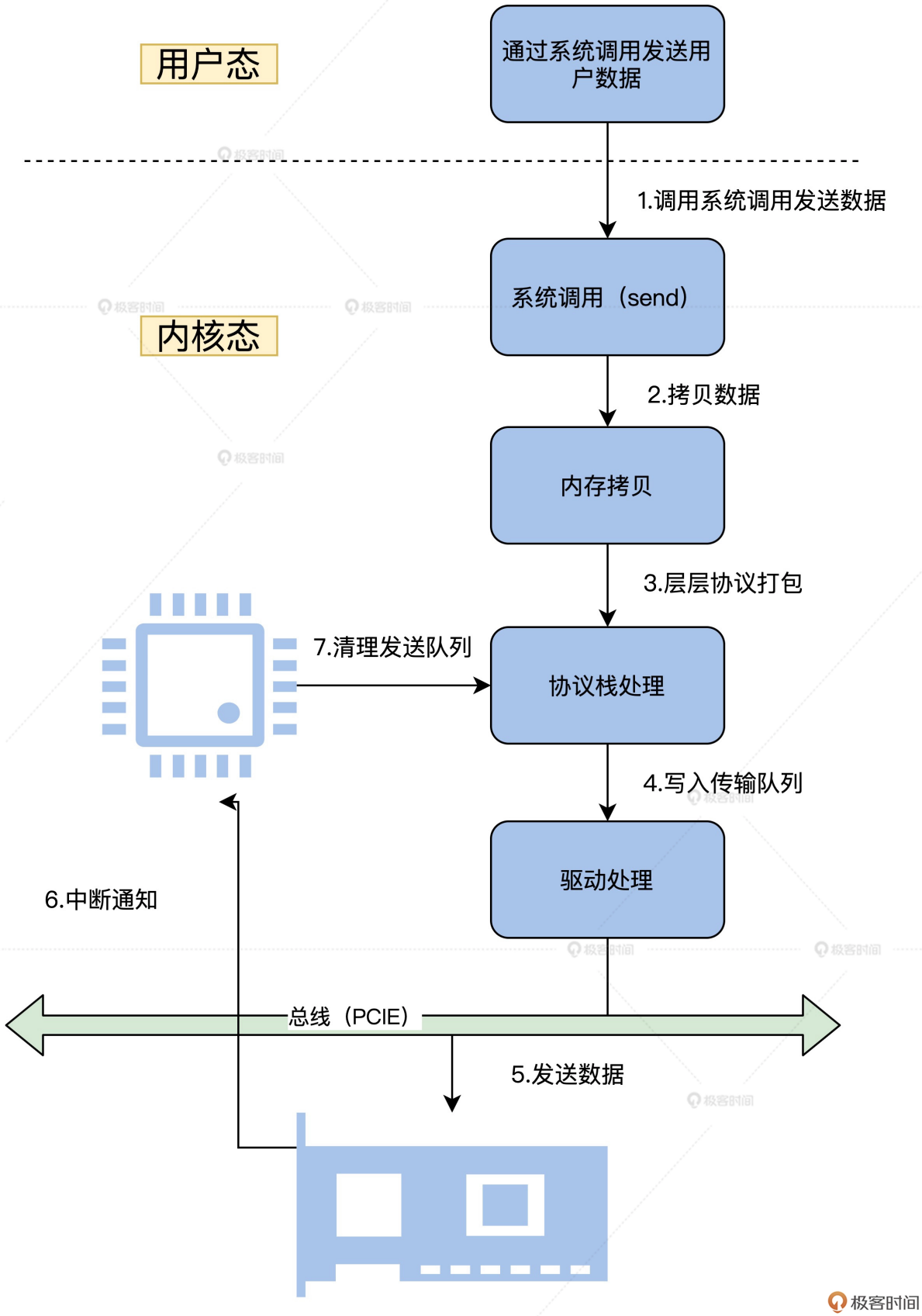
下面我们一起来看看应用程序通过网络发送数据的全过程。

应用程序首先会准备好数据，调用用户态下的库函数。接着调用系统 API 接口函数，进入到内核态。

内核态对应的系统服务函数会复制应用程序的数据到内核的内存空间中，然后将数据移交给网络协议栈，在网络协议栈中将数据层层打包。

最后，包装好的数据会交给网卡驱动，网卡驱动程序负责将打包好的数据写入网卡并让其发送出去。

我为你准备了一张流程图供你参考，如下所示。



发送过程总览

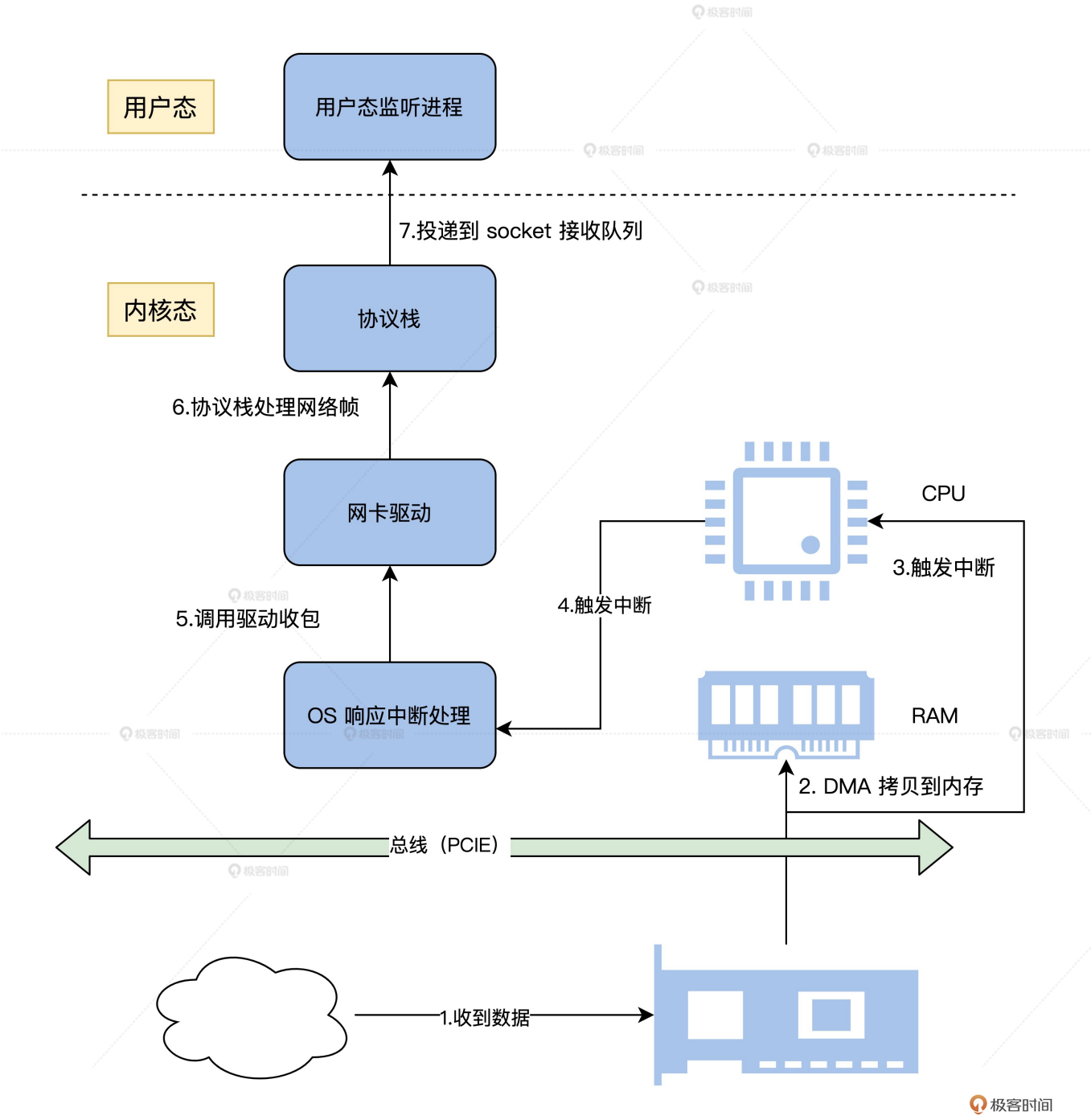
上图中，只是展示了大致流程，其中还有 DMA 处理、CRC 校验、出错处理等细节，但对于我们理解梳理发送流程，这些就够了。

接收过程总览

了解了发送数据的过程以后，掌握接收数据的过程就更容易了，因为它就是**发送数据的逆过程**。

首先，网卡接受到数据，通过 DMA 复制到指定的内存，接着发送中断，以便通知网卡驱动，由网卡驱动处理中断复制数据。然后网络协议收到网卡驱动传过来的数据，层层解包，获取真正的有效数据。最后，这个数据会发送给用户态监听的应用进程。

为了让你更加直观地了解这一过程，我特意准备了一张流程图供你参考，如下所示。



接收过程总览

前面只是帮你梳理一下数据的发送与接收的流程，其实我们真正要关注的是网络协议。可是我们若手动实现一个完整的网络协议，不太现实，网络协议的复杂度大到也许要重新开一门课程，才可以完全解决，所以下面我们借用一下 lwIP 项目，以这个为基础来讨论网络协议。

认识一下 lwIP 架构

现在我们清楚了一次具体网络发收过程是怎么回事，那怎么让 Cosmos 实现网络通信呢？这里我们选择 lwIP 这个 TCP/IP 协议的轻量级开源项目，让它成为 Cosmos 的网络部的战略合作伙伴。

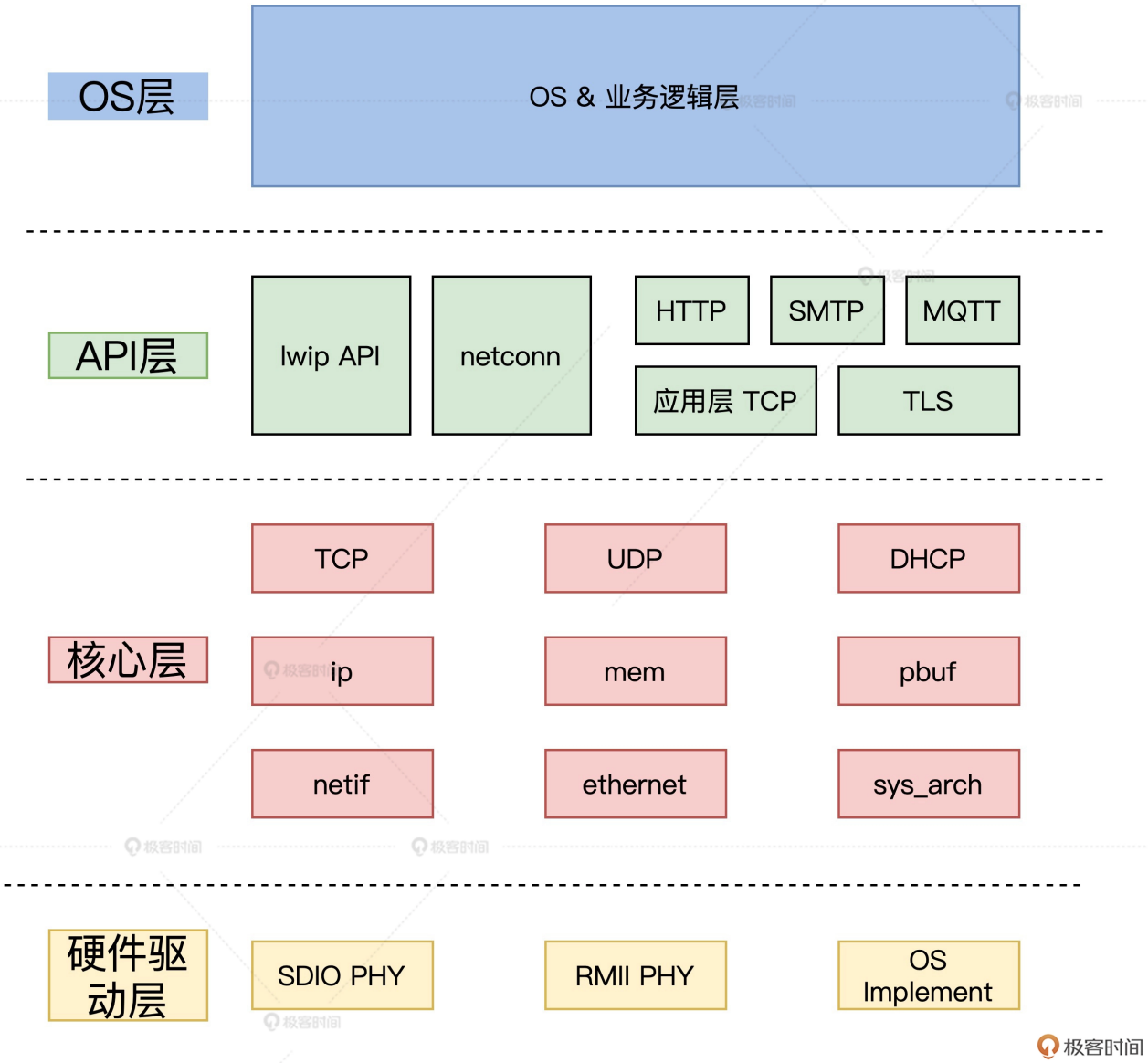
lwIP 是由瑞典计算机科学研究院（SICS）的 Adam Dunkels 开发的小型开源 TCP/IP 协议栈。它是一个用 C 语言实现的软件组件，一共有两套接口层，向下是操作系统要提供的，向上是提供给应用程序的。这样 lwIP 就能嵌入到任何操作系统之中工作，并为这个操作系统上的应用软件提供网络功能支持了。

为啥说 lwIP 是轻量级的呢？很简单，跟 Linux 比，从代码行数上就能看得出。lwIP 的设计目标就是尽量用少量资源消耗，实现一个相对完整的 TCP/IP 协议栈。

这里的“完整性”主要是指 TCP 协议的完整性，实现的关键点就是**在保持 TCP 协议主要功能的基础上减少对 RAM 的占用。**同时，lwIP 还支持 IPv6 的标准实现，这也让我们与现代交换设备的对接变得非常方便。

这里额外提供你一份扩展阅读资料，lwIP 的项目主页链接，这里包含了大量相关资料，感兴趣的同学可以课后深入了解。另外，lwIP 既可以移植到操作系统上运行，也可以在没有操作系统的情况下独立运行。

lwIP 在结构上可分为四层：OS 层、API 层、核心层、硬件驱动层，如下图所示。



lwIP架构图副本

第一层

MCU 的业务层是 lwIP 的服务对象，也是其自身代码使用 lwIP 的地方。大部分时候我们都是从这里入手，通过 netconn 或 lwip_api 使用 lwIP 的各种功能函数。

在典型的 TCP 通信的客户端应用程序中，一般先要通过 netconn_new 创建一个 struct netconn 对象，然后调用 netconn_connect 连接到服务器，并返回成功或失败。成功后，可以调用 netconn_write 向服务器发送数据，也可以调用 netconn_recv 接收数据。最后，关闭连接并通过 netconn_close 释放资源。

第二层

lwIP 的 api 层是 netconn 的功能代码所在的层，负责为上层代码提供 netconn 的 api。习惯使用 socket 的同学也可以使用 lwip_socket 等函数，以标准的 socket 方式调用 lwIP。新版本增加了 http、mqtt 等应用的代码，这些额外的应用对目前的物联网通信来说确实很方便。

第三层

lwIP 的核心层存放了 TCP/IP 协议栈的核心代码，它不仅实现了大部分的 TCP 和 UDP 功能，还实现了 DNS、ICMP、IGMP 等协议，同时也实现了内存管理和网络接口功能。

该层提供了 sys_arch 模块设计，便于将 lwIP 移植到不同的操作系统，如线程创建、信号量、消息队列等功能。和操作系统相关的真正定义写在了 lwip/include/sys.h 文件中。

第四层

硬件驱动层提供 PHY 芯片驱动，用来匹配 lwIP 的使用。lwIP 会调用该层的代码将组装好的数据包发送到网络，同时从网络接收数据包并进行分析，实现通信功能。

lwIP 的三套应用程序编程接口

理清了架构，我们再说一说 lwIP 的应用程序编程接口，一共有三套。

原始 API：原始的 lwIP API。它通过事件回调机制开发应用程序。该应用编程接口提供了最佳的性能和优化的代码长度，但它增加了应用程序开发的复杂性。

Netconn API：是高级的有序 API、需要实时操作系统（RTOS）的支持（提供进程间通信的方法）。Netconn API 支持多线程。

BSD 套接字 API：类似伯克利的套接字 API（在 Netconn API 上开发，需要注意 NETCONN API 即为 Sequential API）。

对于以上三种接口，前者只需要裸机调用，后两种需要操作系统调用。因此，移植 lwIP 有两种方法，一种是只移植内核，不过这样之后只能基于 RAW/Callback API 编写应用程序。第二种是移植内核和上层 API。这时应用程序编程可以使用三种 API，即 RAW/Callback API、顺序 API 和 Socket API。

lwIP 执行流程

现在，想必聪明的你已经理解了前文中的网络收发过程。

接下来，让我们顺着之前的思路来对应到 lwIP 在收发过程中的核心函数，具体过程我同样梳理了流程图。你可以结合图里关键的函数名以及步骤顺序，按这个顺序在 lwIP 代码中检索阅读。

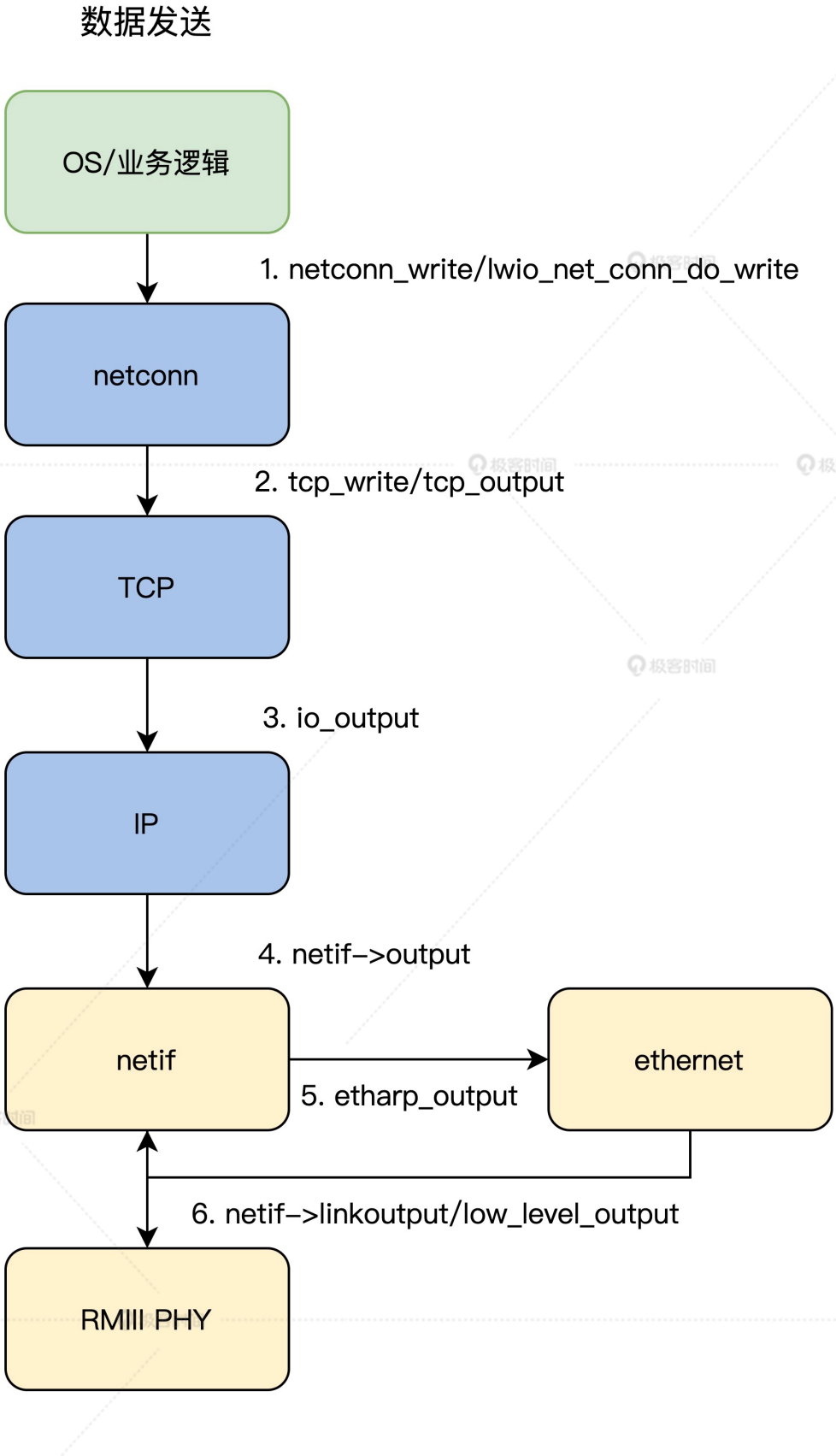
数据发送

首先要说的是数据发送过程。

由于我们把 lwIP 作为 Cosmos 的一个内核组件来工作，自然要由 lwIP 接受来自内核上层发来的数据。内核上层首先会调用 lwIP 的 netconn 层的接口函数 **netconn_write**，通过这个函数，数据正式流进 lwIP 组件层。

接着，netconn 层调用 lwIP 组件的 TCP 层的接口函数 tcp_write，在 TCP 层对数据首次进行打包。然后，TCP 层将打包好的数据通过调用 io_output 函数，向下传递给 lwIP 组件的 IP 层，进行打包。

最后，IP 层将打包好的数据发送给网卡驱动接口层 netif，这里调用了实际的网卡驱动程序，将数据发送出去。


数据发送
OS/业务逻辑
1. netconn_write/lwio_net_conn_do_write
netconn
2. tcp_write/tcp_output
TCP
3. io_output
IP
4. netif->output
netif
ethernet
5. etharp_output
6. netif->linkoutput/low_level_output
RMIII PHY
极客时间

数据发送逻辑

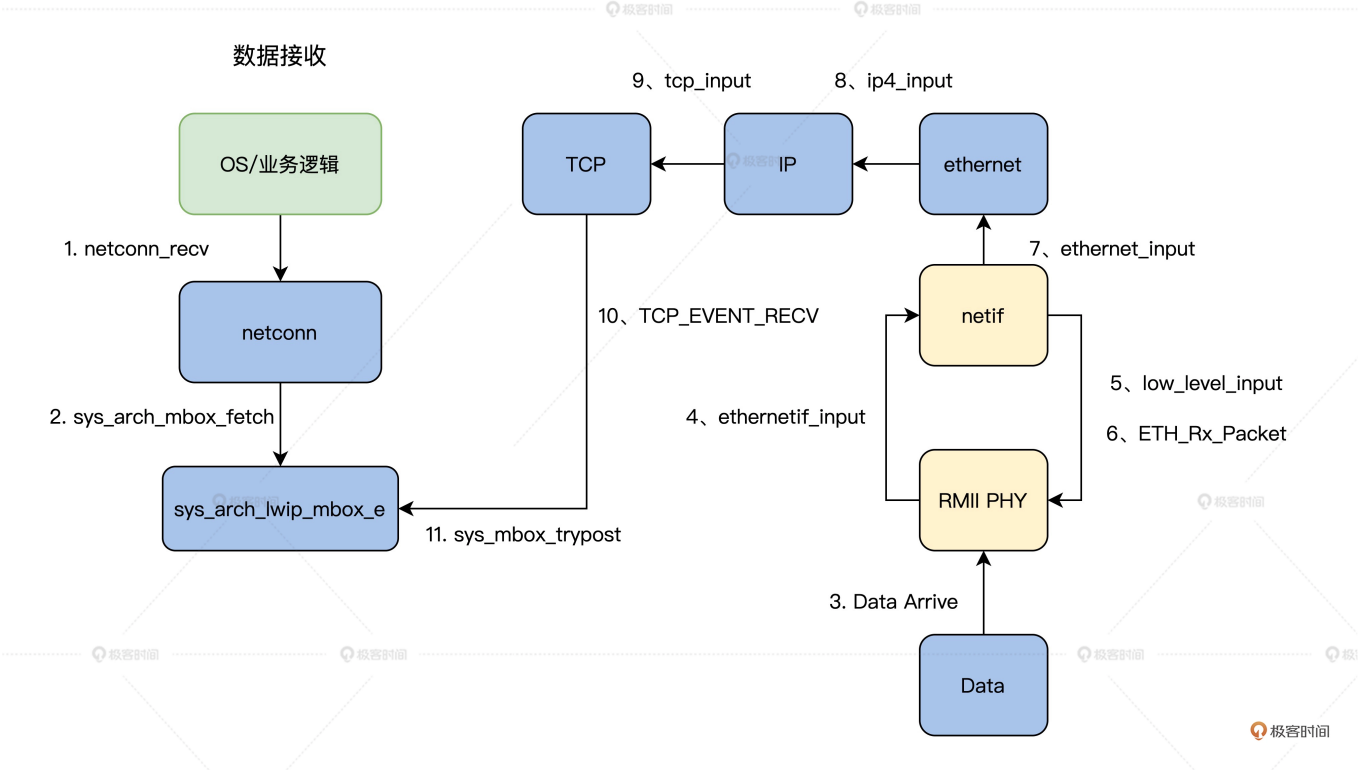
数据接收

数据接收的步骤相比数据发送稍微多一些，但也不用害怕，跟住我的讲解思路一定可以理清这个过程。

数据接收需要应用程序首先调用 lwIP 的 netconn 层的 netconn_recv 接口。然后由 netconn 层调用 sys_arch_mbox_fetch 函数，进入监听等待相关的 mbox。

接着，数据会进入网卡，驱动程序相关的函数负责把它复制到内存。再然后是调用 ethernet_input 函数，进入 ethernet 层。完成相关处理后，调用 ip4_input 函数，数据在 lwIP 组件的 IP 层对数据解包，进行相应处理之后，还会调用 tcp_input 函数，进入 lwIP 组件的 TCP 层对数据解包。

最后，调用 sys_mbox_trypost 函数把数据放入特定的 mbox，也就是消息盒子里，这样等待监听的应用程序就能得到数据了。



数据接收逻辑

在了解了 lwIP 组件收发数据的过程之后，就可以进行移植的相关工作了。lwIP 的结构设计非常优秀，这让移植工作变得很容易。我们这里只要了解 lwIP 组件的 **sys_arch 层的接口函数**即可。

下面我们一起了解 lwIP 的移植细节。

协议栈移植

lwIP 有两种移植模式，一种是 NO_SYS，无操作系统模式，一种是有操作系统模式。用 NO_SYS 模式比较简单，你可以自行探索。

操作系统模式主要需要基于操作系统的 IPC 机制，对网络连接进行了抽象（信号量、邮箱 / 队列、互斥体等机制），从而保证内核与应用层 API 的通讯，这样做的好处是 **lwIP 内核线程可以只负责数据包的 TCP/IP 封装和拆封，而不用进行数据的应用层处理，从而极大地提高系统对网络数据包的处理效率。**

而这些操作系统模拟层的函数主要是在 sys.h 中声明的，我们一般在 sys_arch.c 文件中完成其定义。所以，我们很清楚，带操作系统的移植就是在无操作系统的基础上添加操作系统模拟层。

再接下来我们就看看操作系统模拟层的编写。

有操作系统模式

在之前的课程里我们已经正确实现了 Cosmos 操作系统了，现在我们就可以在 Cosmos 系统提供的 IPC 等机制基础之上，对照 sys.h 文件中声明的函数一一去实现了。

实际工程中完整移植网络栈，需要将后面表格里的这 30 多个函数全部实现。我会带你完成邮箱和系统线程相关的关键部分移植，其他函数的移树思路也大同小异，这里就不一一演示了。

序号	名称	属性	功能
1	sys_sem_t	数据类型	指针类型，指向系统信号量
2	sys_mutex_t	数据类型	指针类型，指向系统互斥量
3	sys_mbox_t	数据类型	指针类型，指向系统邮箱
4	sys_thread_t	数据类型	指针类型，指向系统任务
5	SYS_MBOX_NULL	宏定义	系统邮箱的空值
6	SYS_SEM_NULL	宏定义	系统信号量的空值
7	SYS_MRTEX_NULL	宏定义	系统互斥量的空值
8	sys_now	函数	内核时钟
9	sys_init	函数	初始化系统
10	sys_arch_protect	函数	进入临界段
11	sys_arch_unprotect	函数	退出临界段
12	sys_sem_new	函数	创建一个信号量
13	sys_sem_free	函数	删除一个信号量
14	sys_sem_valid	函数	判断信号量是否有效
15	sys_sem_set_invalid	函数	将信号量设置无效状态
16	sys_arch_sem_wait	函数	等待一个信号量
17	sys_sem_signal	函数	释放一个信号量
18	sys_mutex_new	函数	创建一个互斥量
19	sys_mutex_free	函数	删除一个互斥量
20	sys_mutex_set_invalid	函数	设置互斥量为无效状态
21	sys_mutex_lock	函数	获取一个互斥量
22	sys_mutex_unlock	函数	释放一个互斥量
23	sys_mbox_new	函数	创建一个邮箱
24	sys_mbox_free	函数	删除一个邮箱
25	sys_mbox_valid	函数	判断邮箱是否有效
26	sys_mbox_set_invalid	函数	设置邮箱为无效状态
27	sys_mbox_post	函数	向邮箱发送消息，一直阻塞
28	sys_mbox_trypost	函数	向邮箱发送消息，非阻塞
29	sys_mbox_trypost_fro	函数	在中断中向邮箱发送消息
30	sys_arch_mbox_fetch	函数	从邮箱中获取消息，阻塞
31	sys_arch_mbox_tryfetc	函数	从邮箱中获取消息，非阻塞
32	sys_thread_new	函数	创建一个线程

极客时间

函数表格

从上表中我们可以发现，这些变量和函数主要面向信号量、互斥体和邮箱，包括创建、删除、释放和获取等各种操作，所以我们需要根据操作系统的规定来实现这些函数。

突然看到这么多功能，是不是有点慌？其实不用怕，因为这些功能的实现起来非常简单。首先，我们通过一个例子来看看邮箱功能的实现。

在 lwIP 中，用户代码通过邮箱与协议栈内部交互。邮箱本质上是指向数据的指针。API 将指针传递给内核，内核通过这个指针访问数据，然后进行处理。相反，内核也是通过邮箱将数据传递给用户代码的。

具体代码如下，关键内容我都做了详细注释。

复制代码

```
/*创建一个空的邮箱。*/
err_t sys_mbox_new(sys_mbox_t *mbox, int size)
{
osMessageQDef(QUEUE, size, void *);
*mbox = osMessageCreate(osMessageQ(QUEUE), NULL);
#if SYS_STATS
++lwip_stats.sys.mbox.used;
if (lwip_stats.sys.mbox.max < lwip_stats.sys.mbox.used) {
lwip_stats.sys.mbox.max = lwip_stats.sys.mbox.used;
}
#endif /* SYS_STATS */
if (*mbox == NULL)
return ERR_MEM;
return ERR_OK;
}
/*重新分配一个邮箱。如果邮箱被释放时，邮箱中仍有消息，在lwIP中这是出现编码错误的指示，并通知
void sys_mbox_free(sys_mbox_t *mbox)
{
if( osMessageWaiting(*mbox) )
{
portNOP();
#if SYS_STATS
lwip_stats.sys.mbox.err++;
#endif /* SYS_STATS */
}
osMessageDelete(*mbox);
#if SYS_STATS
--lwip_stats.sys.mbox.used;
#endif /* SYS_STATS */
}
/*发送消息到邮箱*/
void sys_mbox_post(sys_mbox_t *mbox, void *data)
{
while(osMessagePut(*mbox, (uint32_t)data, osWaitForever) != osOK);
}
/*尝试将消息发送到邮箱*/
err_t sys_mbox_trypost(sys_mbox_t *mbox, void *msg)
```

```
{
err_t result;
if ( osMessagePut(*mbox, (uint32_t)msg, 0) == osOK)
{
result = ERR_OK;
}
else {
result = ERR_MEM;
#if SYS_STATS
lwip_stats.sys.mbox.err++;
#endif /* SYS_STATS */
}
return result;
}
/*阻塞进程从邮箱获取消息*/
u32_t sys_arch_mbox_fetch(sys_mbox_t *mbox, void **msg, u32_t timeout)
{
osEvent event;
uint32_t starttime = osKernelSysTick();;
if(timeout != 0)
{
event = osMessageGet (*mbox, timeout);
if(event.status == osEventMessage)
{
*msg = (void *)event.value.v;
return (osKernelSysTick() - starttime);
}
else
{
return SYS_ARCH_TIMEOUT;
}
}
else
{
event = osMessageGet (*mbox, osWaitForever);
*msg = (void *)event.value.v;
return (osKernelSysTick() - starttime);
}
}
/*尝试从邮箱获取消息*/
u32_t sys_arch_mbox_tryfetch(sys_mbox_t *mbox, void **msg)
{
osEvent event;
event = osMessageGet (*mbox, 0);
if(event.status == osEventMessage)
{
*msg = (void *)event.value.v;
return ERR_OK;
}
else
{
return SYS_MBOX_EMPTY;
```

```
}
}
/*判断一个邮箱是否有效*/
int sys_mbox_valid(sys_mbox_t *mbox)
{
if (*mbox == SYS_MBOX_NULL)
return 0;
else
return 1;
}
/*设置一个邮箱无效*/
void sys_mbox_set_invalid(sys_mbox_t *mbox)
{
*mbox = SYS_MBOX_NULL;
}
// 创建一个新的信号量。而 "count"参数指示该信号量的初始状态
err_t sys_sem_new(sys_sem_t *sem, u8_t count)
{
osSemaphoreDef(SEM);
*sem = osSemaphoreCreate (osSemaphore(SEM), 1);
if(*sem == NULL)
{
#if SYS_STATS
++lwip_stats.sys.sem.err;
#endif /* SYS_STATS */
return ERR_MEM;
}
if(count == 0) // Means it can't be taken
{
osSemaphoreWait(*sem,0);
}
#if SYS_STATS
++lwip_stats.sys.sem.used;
if (lwip_stats.sys.sem.max < lwip_stats.sys.sem.used) {
lwip_stats.sys.sem.max = lwip_stats.sys.sem.used;
}
#endif /* SYS_STATS */
return ERR_OK;
}
```

此外还有一些函数也是协议栈需要的函数，特别是 sys_thread_new 函数，不但协议栈在初始化时需要用到，在后续我们实现各类基于 lwIP 的应用时也会用得到，它的具体实现如下。

复制代码

```
sys_thread_t sys_thread_new(const char *name, lwip_thread_fn thread , void *ar
{
const osThreadDef_t os_thread_def = { (char *)name, (os_pthread)thread, (osPri
```

```
4  return osThreadCreate(&os_thread_def, arg);
5  }
6  osThreadId osThreadCreate (const osThreadDef_t *thread_def, void *argument)
7  {
8  TaskHandle_t handle;
9  #if( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCAT
10 if((thread_def->buffer != NULL) && (thread_def->controlblock != NULL)) {
11 handle = xTaskCreateStatic((TaskFunction_t)thread_def->pthread,(const portCHAR
12 thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
13 thread_def->buffer, thread_def->controlblock);
14 }
15 else {
16 if (xTaskCreate((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_d
17 thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
18 &handle) != pdPASS) {
19 return NULL;
20 }
21 }
22 #elif( configSUPPORT_STATIC_ALLOCATION == 1 )
23 handle = xTaskCreateStatic((TaskFunction_t)thread_def->pthread,(const portCHAR
24 thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
25 thread_def->buffer, thread_def->controlblock);
26 #else
27 if (xTaskCreate((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_d
28 thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
29 &handle) != pdPASS) {
30 return NULL;
31 }
32 #endif
33 return handle;
34 }
```

至此，基于 Cosmos 操作系统移植 lwIP 协议栈的关键部分就算完成了。

重点回顾

好，这节课的内容告一段落了，我来给你做个总结。

我们首先从数据发送接收的视角，观察了数据从用户态到内核态，再从内核态到流动到用户态的全过程。

接着，我们发现网络协议栈移植与 DMA、内核的 IPC、信号量、DMA 等机制密切相关。理解网络栈移植的关键步骤，能够让我们更好地理解内核特性在工程中是如何应用的。

最后，我们实现了将 lwIP 网络协议栈的关键部分移植到 Cosmos 操作系统下。不过这节课我带你实现了邮箱和系统线程相关的关键部分，其他函数移植道理相通，感兴趣的同学可以自行探索。

思考题

我们已经了解到了操作系统内核和网络协议栈的关系，可是网络协议栈真的一定只能放在内核态实现么？

欢迎你在留言区跟我交流探讨。也欢迎你把这节课分享给自己的朋友、同事。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 36 | 从URL到网卡：如何全局观察网络数据流动？

下一篇 38 | 从单排到团战：详解操作系统的宏观网络架构

小争哥新书

数据结构与算法之美

图书+专栏，双管齐下，拿下算法

打包价 ¥159 原价¥319

仅限 300 套

精选留言 (6)

写留言

相逢是缘 置顶
2021-08-02

我们的 Cosmos 操作系统已经实现了xTaskCreateStatic、osSemaphoreCreate 等这些函数了吗？
一路跟着课程过来，没有发现呢，37课代码里面也没有呢

> 编辑回复: 相逢是缘你好，感谢你的留言。需要自己在Cosmos实现 ，这里代码只是为了演示移植的思路。先理解移植的核心思想即可，暂时不必深究。受限于工程规模，Cosmos系统的网络栈移植将会安排在项目正式开源之后排期开发，同学们敬请期待哈～

1

pedro
2021-08-02

由于历史的原因，主流的操作系统都是在内核实现网络协议栈，但是由于网络环境的日益复杂，用户对服务稳定性的要求越来越高。
内核网络协议栈的封闭性是一个很大的桎梏点，因此非内核网络协议栈早就被很多人实现

了，例如 DPDK，号称能解决 c10m 问题。虽然现在内核协议栈仍然是主流，但可以预见的是非内核协议栈会越来被人接受，解决更多的实际需求。

展开

作者回复: 期待老铁能够实现Cosmos用户待协议栈

1

Fan
2021-08-03

找到个 lwIP的教程。
https://www.kancloud.cn/jiejietop/tcpip/988547

编辑回复: 感谢分享~

1

青玉白露
2021-08-02

网络协议栈也可以放在用户态来执行。
需要注意的是，如今网速加快、网络环境复杂，多核的CPU在处理高速的网络请求时也会有很大的性能局限性。
而将网络协议栈放在用户态，就可以实现高度自定义，有关这方面的知识可以参考这个博客：https://www.cnblogs.com/jmilkfan-fanguiju/p/12789805.html

展开

作者回复: 6666

1

LDxy
2021-08-02

协议栈应该也可以放在用户态实现，这时候就需要绕过内核的协议栈，直接和网卡驱动程序通信

作者回复: 是的

云师兄

2021-08-02

理解软件的设计思想，最重要的是先要理解需求。而内核中的数据流转也只是为了满足网络收发的需求而进行的设计。

展开

作者回复: 是的